

CIRCULATION COPY
SUBJECT TO RECALL
IN TWO WEEKS

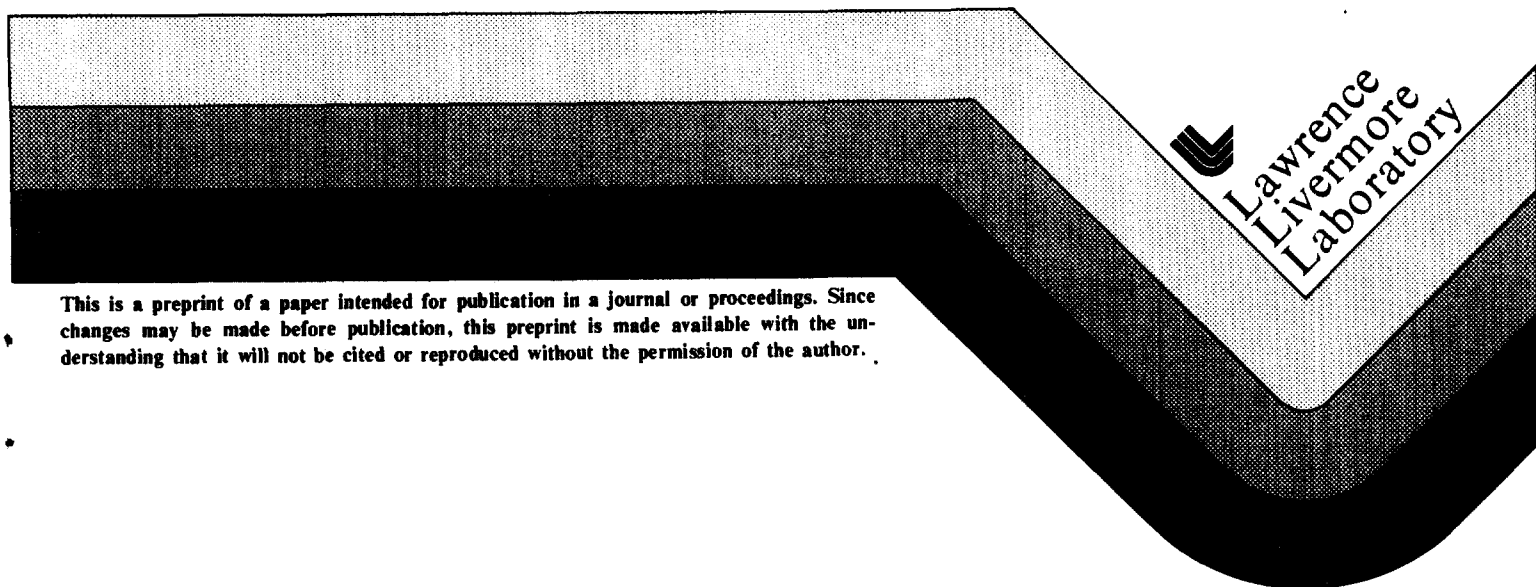
UCRL-83674
PREPRINT

2-D FFTS OF LARGE IMAGES WITH THE AP-120B

Richard E. Twogood

This paper was prepared for submittal to the
1980 Floating Point Systems Users Group Meeting,
San Francisco, California, April 1980

February 29, 1980



DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

2-D FFTs OF LARGE IMAGES WITH THE AP-120B

Richard E. Twogood

Lawrence Livermore Laboratory

Livermore, CA

Abstract

This paper investigates the issues involved in implementing a 2-D FFT on the FPS AP-120B array processor when the data memory available is less than the image size. After a brief review of the alternative techniques that have been proposed in the literature (with matrix transposition, without matrix transposition, vector radix), a recently developed "two-level" implementation is described. An analysis of the CPU and I/O requirements is given, showing that this algorithm is significantly superior to existing methods due to the reduced I/O requirements.

Introduction

As in standard 1-D signal processing applications, the Fast Fourier Transform (FFT) plays a crucial role in 2-D digital signal processing. Examples of applications include nonrecursive linear filtering, some nonlinear filtering techniques, 2-D spectral analysis, and image restoration. As a result, a number of efficient techniques for implementing a 2-D FFT of disk-based images

have been proposed [1]-[4]. None of these techniques is substantially better than the others in most practical implementations due to the similarity of their input/output (I/O) structures. Recently, however, Ari [5] proposed what he calls a two-level matrix transposition algorithm with a particularly efficient I/O structure. When applied to the problem of implementing 2-D FFT's of images larger than the available primary memory, this technique gives an implementation which is superior, in general, to any other method proposed in the literature.

In this paper, we briefly review the alternative 2-D FFT algorithms and discuss how the new two-level algorithm gains its advantage. An analysis of the implementation of this algorithm on the FPS AP-120B array processor is given, followed by examples which illustrate the benefits derived by this approach.

The analysis in this paper addresses system configurations where there is direct transfer capability between the array processor and the disk on which the image is stored. This situation exists, of course, when a dedicated disk is interfaced through a GPIOP into the array processor. At LLL, we have accomplished the same effect by implementing software that allows for direct "device to device" transfers without passing data through the host computer.

2-D FFT of Disk-based Images

The historical approach for computing 2-D FFT's of arrays larger than primary storage consists of decomposing the 2-D FFT into iterated 1-D FFT's. The

calculation proceeds as follows: first each row in the array is transformed with the 1-D FFT. The array is then transposed and the row transform process is repeated (this computes the column transforms of the original array). This three-step procedure yields the transpose of the transformed array.

A significant part of the data manipulation required by the above procedure is a result of the matrix transpositions. A fast transposition scheme for images of dimension $2^n \times 2^n$ was proposed by Eklundh in [1]. Eklundh's method, based on the properties of the binary address expansion of the elements of the matrix before and after transposition, consists of n steps. In the k^{th} step ($1 \leq k \leq n$) every other vector of length 2^{k-1} is permuted with another vector of the same length, i.e. exactly one-half of the matrix elements are permuted in each step. The element interchanges are performed between elements in rows spaced 2^{k-1} rows apart. As a result, pairs of adjacent rows are required in the first step, pairs of rows spaced 2 apart are required in the second step, pairs of rows spaced 4 apart are needed in the third step, etc. If one can fit 2^j of the 2^n image rows in the computer's memory ($1 \leq j \leq n$), it is easily seen that $\left\lceil \frac{n}{j} \right\rceil$ I/O passes are required to complete the transposition, where the notation $\lceil x \rceil$ denotes the smallest integer greater than or equal to x . It is also readily seen that the first step requires the fewest disk accesses, as 2^j contiguous rows can be accessed at once to obtain the data required for the first j steps. On the other hand, the intermediate I/O accesses require single line accesses (2^n reads and 2^n writes of disk) to obtain the necessary data. The final I/O pass requires $2^{2(n-j)} - n'j$ reads and the same number of writes of disk, where $n' = \left\lceil \frac{n}{j} \right\rceil - 2$ is the number of

intermediate I/O passes. The Eklundh transposition scheme, then, can be characterized by the following timing equations:

$$\begin{aligned} T_{\text{CPU-EK}} &\approx n 2^{2n-2} T_{\text{change}} \\ T_{\text{I/O-EK}} &\approx (2^{n-j+1} + n' 2^{n+1} + 2^{2(n-j)-n'j+1}) T_{\text{acc}} + \\ &\quad \left[\frac{n}{j} \right] 2^{2n+1} T_{\text{tran}} \end{aligned} \quad (1)$$

where T_{change} is the average time to permute two elements in the data memory, T_{acc} is the access time of the disk, and T_{tran} is the transfer rate.

Twogood and Ekstrom [2] extended the Eklundh transposition scheme by combining element permutations in the case when more than 2 rows fit into the memory ($j > 1$). While this resulted in an improvement in CPU time, the I/O structure was identical to that of Eklundh's technique.

Two alternative schemes for computing the 2-D FFT without a matrix transposition have been developed in [3] and [4]. Onoe's method [3] simply involves performing the row FFT's, and then computing individual stages of the column FFT's by performing the appropriate I/O. Harris' method [4] involves the same idea of accessing sets of rows to allow individual stages of the column FFT's to be performed, but it also combines the row FFT's and column FFT's into truly 2-D FFT "butterflies", rather than computing iterated 1-D

transforms. Careful examination of those algorithms, however, shows that they require exactly the same I/O procedure as Eklundh's matrix transposition scheme. This is a result of the similarity between data requirements for the matrix transposition and the column FFT, i.e. column elements spaced by successive powers of 2 are required by either the matrix transposition or the column FFT.

As an example, consider the case $n=5$ and $j=2$, i.e. we have an image that is 32×32 and only 4 rows of the image fit into the memory. The I/O structure for this case is illustrated in Figure 1; the 32 rows are represented in the vertical direction and the $\left\lceil \frac{n}{j} \right\rceil = 3$ required I/O passes in the horizontal direction. Note that 4 contiguous rows are read at a time in the first pass, but sets of 4 single line transputs are required in the second pass. The third and final pass requires transputs of 2 contiguous lines at a time.

Until recently, the I/O scheme illustrated by the above example was the best available one, and was incorporated into each of the 2-D FFT algorithms of [1]- [4]. A new "two-level" algorithm was proposed in [5], however, which offers significant improvement in the I/O for all cases where $\left\lceil \frac{n}{j} \right\rceil > 2$. This technique is based on the usage of a relatively small, in general, scratch storage on the disk being used. It gains its I/O advantage by writing the results of the intermediate passes in contiguous chunks to the scratch store. Thus, while 2^n reads of one row each are still required in the intermediate passes, only 2^{n-j} writes of 2^j rows are required for those passes. While the CPU requirements are unchanged for this technique (identical to the

extended-Eklundh method), the I/O times are significantly improved, with

$$T_{I/O-2L} \approx \left(2^{n-j+1} + n' \left(2^n + 2^{2(n-j)-n'j} \right) \right) T_{acc} \quad (2)$$

$$+ 2^{2n+1} (n'+2) T_{tran} \quad n' > 0$$

$$\left(2^{n-j+1} + 2^{2(n-j)+1} \right) T_{acc} + 2^{2n+2} T_{tran} \quad n'=0$$

where $n' = \left\lceil \frac{n}{j} \right\rceil - 2$ is once again the number of intermediate I/O passes. We note that Equ. (2) above agrees with the results of Table II in [5], except for the case $n'=0$. For that case, the results of [5] are incorrect as they ignore the fact that the write accesses need not necessarily be done for individual lines on the final pass.

PASS	1	2	3
0	1	1	1
4	2	2	2
8	3	3	3
12	4	4	4
16	5	5	5
20	6	6	6
24	7	7	7
28	8	8	8

Figure 1. Eklundh I/O Structure

PASS	R/O	W/O
0	1	1
4	2	2
8	3	3
12	4	4
16	5	5
20	6	6
24	7	7
28	8	8

Scratch Store	Write Only	Read Only
0	1,3,5,7	1,3,5,7
4	2,4,6,8	2,4,6,8

Figure 2. Two-level Structure

To better understand the I/O structure of this two-level algorithm, consider the same $n=5$ and $j=2$ example as before. A layout of the I/O required is given in Figure 2. Note that there is but one intermediate stage which uses the scratch store, and 8 lines of scratch store are required. In general, this

technique requires $\sum_{k=1}^{n'} 2^{n-kj}$ lines of scratch storage, as shown in [5].

FFT timing: With the above results, we are now in a position to characterize the total computational requirements of the 2-D FFT. Assuming a real input image, we require $N=2^n$ real FFT's of length N on the rows. After transposition, $\frac{N}{2} - 1$ complex FFT's and 2 real FFT's of length N are performed on the columns. In addition, the array processor CPU time to perform the required vector swaps must be accounted for. As shown in [2], the CPU time for the matrix transposition is

$$T_{XP} \approx \frac{1}{2} (n'+1) (2^{2n} - 2^{2n-j}) T_{\text{change}} + \frac{1}{2} (2^{2n} - 2^{n+jn'+j}) T_{\text{change}} \quad (3)$$

where T_{change} is again the average time to permute 2 elements in the data memory. The total CPU time is therefore

$$T_{\text{CPU-2L}} \approx (N+2) T_{\text{RFFTN}} + \left(\frac{N}{2}-1\right) T_{\text{CFFTN}} + T_{XP} \quad (4)$$

where T_{RFFTN} and T_{CFFTN} are the AP times to perform real and complex N -length FFT's, respectively. The total FFT time is therefore

$$T_{2\text{DFFT}} \approx T_{\text{CPU-2L}} + T_{\text{I/O-2L}} \quad (5)$$

This approximate timing will, of course, hold only if the AP calls are effectively "vector function chained" and if the host CPU performs its disk address calculations, loop indexing, and control functions efficiently.

A few example analyses of some practical image processing applications are given in Figure 3. In the examples, we have assumed an FPS AP-120B with 167ns memory, and assumed a CDC 9762 with $T_{acc} \approx 20$ ms and $T_{tran} \approx 4\mu s$.

n	j	Lines Transferred	Accesses	T_{CPU-2L}	$T_{I/O-2L}$	T_{2DFFT}
8	3	1536	736	.47 sec	16.3 sec	16.8 sec
9	4	3072	1184	2.0 sec	30.0 sec	32.0 sec
9	5	2048	544	1.8 sec	15.1 sec	16.9 sec
10	4	6144	2496	8.3 sec	75.1 sec	83.4 sec
10	5	4096	2112	7.7 sec	59.0 sec	66.7 sec
11	3	16384	8192	37.6 sec	298 sec	336 sec
11	4	12288	5504	35.0 sec	210 sec	245 sec
11	5	12288	4416	35.0 sec	189 sec	224 sec

Figure 3. Example 2-D FFT Timings Using Two-Level Algorithm

As a comparison, consider the (n=11, j=3) case, which corresponds to a 2048x2048 image to be transformed and only 16K of data memory. As the table shows, 8192 accesses are needed and the timing estimate is 336 seconds. The extended Eklundh method, on the other hand, would require 10752 accesses and approximately 388 seconds.

While the above results are approximate predictions, we have achieved fairly good correlation with the actual timings of the 2-D FFT program implemented at LLL. For 512x512 images with 16K of data memory ($n=9$, $j=5$), we have timed the 2-D FFT at 18 seconds, which agrees closely with the 16.9 seconds predicted. The largest discrepancies are caused by the relatively costly address calculations and control performed in the host PDP 11/70.

Conclusions

Some of the issues involved in the implementation of large 2-D FFT's on memory-limited processors have been discussed. A brief comparison of existing techniques for 2-D FFT's was given, followed by an analysis of the benefits derived by using a new two-level I/O structure for the implementation of matrix transpositions. The implementation of this two-level algorithm for the 2-D FFT with an FPS AP-120B array processor was discussed, with timing analyses and examples for image processing applications of practical interest.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

References

- (1) J. O. Eklundh, "A Fast Computer Method for Matrix Transposing," IEEE Trans. Comp., Vol. C-21, July 1972, pp. 801-803.
- (2) R. E. Twogood and M. P. Ekstrom, "An Extension of Eklundh's Matrix Transposition Algorithm and Its Application in Digital Image Processing," IEEE Trans. Comp., Vol. C-25, September 1976, pp. 950-952.
- (3) M. Onoe, "A Method for Computing Large-Scale Two-Dimensional Transform Without Transposing Data Matrix," Proc. IEEE, Vol. 63, No. 1, January 1975, pp. 196-197.
- (4) D. B. Harris, et al., "Vector Radix Fast Fourier Transform," Proc. 1977 IEEE Inter. Conf. on Acoustics, Speech, and Signal Processing, Hartford, Connecticut, May 1977, pp. 548-551.
- (5) M. B. Ari, "On Transposing Large $2^n \times 2^n$ Matrices," IEEE Trans. Comp., Vol. C-27, January 1979, pp. 72-75.